How to Find the Exit from a 3-Dimensional Maze*

Miki Hermann
LIX, CNRS, École Polytechnique, Institut Polytechnique de Paris, 91120 Palaiseau, France

Abstract

We present several experimental algorithms for fast computation of variadic polynomials over non-negative integers.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases Young tableaux, randomized algorithm, probabilistic algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.21

Supplementary Material *Software (Source Code & Data)*: `https://github.com/miki-hermann/gyt` archived at `swh:1:dir:aa547a4af49a9563138a13168af0160d9e709954`

1 Introduction and Motivation

Imagine a three-dimensional cubic maze structure called the *Cube*. Each side of the *Cube* spans 26 rooms and there are $26 \times 26 \times 26 = 17576$ rooms in total. Except for the rooms on the edges or faces of the *Cube*, each room has 6 neighbors: up, down, left, right, front, and back. Each room is identified by its coordinates x, y, and z, ranging from 0 to 25. Moreover, each room has a label written on its floor, determined by an unknown ternary function $f\colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ over natural numbers. The parameters of the function f are the coordinates of the room. The only information you have about the function f is that it is increasing in each coordinate, i.e., that the following relations hold

$$f(x,y,z) < f(x+1,y,z), \quad f(x,y,z) < f(x,y+1,z), \quad f(x,y,z) < f(x,y,z+1)$$

for each coordinate (x,y,z). You do not know the labels of the rooms upfront, but discover them by visiting the rooms on your path to the exit. Labels are not unique: two different rooms can have the same label. You can pass from one room to another if there exists a door between them. Each pair of neighboring rooms share a door, which means that there is a door between any two rooms sharing a face. Except for the rooms on the edges and outer faces of the *Cube*, from a given room you can pass to a neighbor room up, down, left, right, front, or back. Formally speaking, from a room with coordinates (x,y,z) you can pass to one of the rooms with coordinates

$$(x+1,y,z), \quad (x,y+1,z), \quad (x,y,z+1), \quad (x-1,y,z), \quad (x,y-1,z), \quad (x,y,z-1),$$

when $0 < x,y,z < 25$. Contrary to the movie, there are no deadly traps in the rooms. Nevertheless, you are not allowed to pass between rooms freely. You cannot return back to a previously visited room unless you have flagged it. If you are not sure which choice to make, you can flag the current room, so that you can return to it later. If you decide in a certain moment that you arrived at a dead-end, you can ask to be teleported back to the last flagged room. You can return to each flagged room only once, i.e., you have two choices to move from a flagged room to another room, allowing you a limited backtrack, contrary to unflagged rooms where you have only one choice. You have 29 flags available, i.e., you have

* Inspired by the Horror Movie *Cube*.

the possibility to return to 29 branchings. Once a room is flagged, you cannot remove it any more. The coordinates of the flagged rooms are maintained in a stack. You can return only to the room whose flag is on top of the stack. Once you return to a flagged room, its coordinates are popped from the stack. The exit room is labeled by 131350013988347832235. Your starting position is the room with coordinates $(0, 0, 25)$ labeled by 162981450557708740234375. Are you able to find the exit? What is the minimal number of rooms you must pass through from your starting position to the exit?[1]

2 Analysis

Before passing to the three- and more-dimensional case, let us analyze the problem in lower dimensions.

2.1 Linear Board

In one dimension, the analysis is quite easy. We have a linear board of length n with coordinates $0, \ldots, n-1$, an unknown unary function $f\colon \mathbb{N} \to \mathbb{N}$, a starting position s, and an exit label B. For two different positions $a, b \in \{0, \ldots, n-1\}$ on board we know that $a < b$ implies $f(a) < f(b)$. Hence the exit label B can occur only once on a linear board. The starting point s is one of the extremities of the board: either $s = 0$ or $s = n-1$.

The search algorithm proceeds as follows. First, we set $x \leftarrow s$ and compute the value $f(x)$. If $f(x) = B$ holds, then we are already at the exit room. If $f(x) > B$ we must decrease x, else if $f(x) < B$ we must increase x. Set $x \leftarrow x-1$ or $x \leftarrow x+1$, respectively, and repeat the loop. No flags are necessary to reach the exit, since there is no necessity to make choices.

In the worst case, the starting point is at one extremity of the board (say 0), and the exit at the other $(n-1)$. Hence the path to the exit must contain n rooms in the worst case. On average, each position from $0, \ldots, n-1$ is equally likely to contain the exit. The probability p_i that a position i contains the exit is $p = 1/n$. We denote by X the random variable equal to the length of the path from 0 to the exit and set $\Pr[X = i] = p_{i-1} = 1/n$. If the starting point is one of the extremities ($s = 0$ or $s = n-1$, but these two cases are mirror images of each other), the expected length of the path to the exit is

$$E[X|s=0] \;=\; E[X|s=n-1] \;=\; \sum_{i=1}^{n} i \cdot \Pr[X=i] \;=\; \sum_{i=1}^{n} \left(i \cdot \frac{1}{n} \right) \;=\; (n+1)/2.$$

Not really a surprise, this position is near the middle of the linear board.

2.2 Matrix Board

A $m \times n$ matrix A, whose elements $A[x, y]$ are equal to a binary function $f(x, y)\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, satisfying the relations$f(x, y) < f(x+1, y)$ and $f(x, y) < f(x, y+1)$, is a full Young tableau over natural numbers [2, Problem 6-3, page 143]. The starting position s is usually $(0, 0)$, but for simplification reasons we will consider the coordinate $(0, n-1)$ as the starting point. Just consider the matrix horizontally flipped. For two different positions $a = (a_1, a_2)$ and $b = (b_1, b_2)$ there exist two positions $c = (c_1, c_2)$ and $d = (d_1, d_2)$, such that $c_i = \min\{a_i, b_i\}$

[1] Just for your information, the exit is located in the room with coordinates $(14, 15, 16)$ and the minimal number of visited rooms is therefore 39, including the starting room and the exit.

Algorithm 1 Search in a 2D Maze.

Input: Function $f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ satisfying $f(x,y)<f(x+1,y)$ and $f(x,y)<f(x,y+1)$, and a value $B\in\mathbb{N}$.
Output: Coordinates (x,y) for which $f(x,y)=B$ or $\perp$ if such coordinates do not exist.

```
1:  function 2D_SEARCH(f, B)
2:      x ← 0
3:      y ← n − 1
4:      while x ≤ m − 1 & y ≥ 0 do
5:          if f(x, y) = B then
6:              return (x, y)
7:          else if f(x, y) < B then
8:              x ← x + 1
9:          else if f(x, y) > B then
10:             y ← y − 1
11:         end if
12:     end while
13:     return ⊥
14: end function
```

and $d_i=\max\{a_i,b_i\}$, satisfying the relations

$$f(c_1,c_2)\le f(a_1,a_2), f(c_1,c_2)\le f(b_1,b_2), f(a_1,a_2)\le f(d_1,d_2), \text{ and } f(b_1,b_2)\le f(d_1,d_2).$$

All four relations are strict if the positions a and b do not share the same row or column. Hence, the exit label B can occur only once in each row and only once in each column.

Algorithm 1 proceeds as follows. The starting position is the room s with coordinates $(0,n-1)$, therefore we set $x\leftarrow 0$ and $y\leftarrow n-1$. While $f(x,y)>B$ holds, decrease the second coordinate: $y\leftarrow y-1$. When we arrive at a position where $f(x,y)<B$, we increase the first coordinate: $x\leftarrow x+1$. We repeat this loop until we find a room labeled by B. No flags are necessary to reach the exit, since there is no necessity to make choices.

The correctness of Algorithm 1 is easily proved. If $f(x,y)>B$ holds, then we have $f(x',y)>B$ for each $x'>x$. Hence the exit room labeled by B cannot be located at any position (x',y') for $x'\ge x$ and $y'\ge y$. Therefore there is no need to increase the first coordinate if the $f(x,y)>B$ holds. Only the second coordinate can be decreased to move towards the exit room. If $f(x,y)<B$ holds, then we have $f(x,y')<B$ for each position $y'<y$. Hence the exit room labeled by B cannot be located at any position (x',y') for $x'\le x$ and $y'\le y$. Therefore there is no need to decrease the second coordinate. Only the first coordinate can be increased to move towards the exit room.

In the worst case, the starting point is at the south-east extremity $(0,n-1)$ of the maze and the exit at the north-western extremity $(m-1,0)$. Algorithm 1 proceeds by a zig-zag, which never returns back. Neither the coordinate y (columns) is increased, nor the coordinate x (rows) is decreased. There are m rooms in each row and n rooms in each column. Therefore the path to the exit must contain $m+n$ rooms in the worst case. This is a considerable improvement against a brute force algorithm going through all $m\cdot n$ rooms.

3 The *Cube* and Beyond

Let us extend the previous ideas to a cubic maze. We have $\ell\times m\times n$ cube A, whose elements $A[x,y,z]$ are equal to a ternary function $f(x,y,z)\colon \mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{N}$, satisfying the inequalities

$f(x,y,z) < f(x+1,y,z)$, $f(x,y,z) < f(x,y+1,z)$, and $f(x,y,z) < f(x,y,z+1)$. The starting position is, once again, $(0,0,n-1)$. If you wish to start at the origin $(0,0,0)$, you can just flip the cube and arrange the subsequent computation according to this flip.

The basic idea of the algorithm remains the same. We set $x \leftarrow 0$, $y \leftarrow 0$, and $z \leftarrow n-1$ at the beginning. While $f(x,y,z) > B$ holds, we decrease the last coordinate: $z \leftarrow z-1$. This is correct, because we have $f(x',y',z) > B$ for any $x' \geq x$ and $y' \geq y$ when $f(x,y,z) > B$ holds. This implies that the value B cannot be in the cube slice with the fixed z. When we reach a coordinate z with $f(x,y,z) < B$, then we have $f(x',y',z') < B$ for any position with $x' \leq x$, $y' \leq y$, and $z' \leq z$. However, we now have the choice to increase either x or y, contrary to the two-dimensional case, where we were forced to increase only one variable. Here, we have the choice to continue either to the room $(x+1,y,z)$ or $(x,y+1,z)$. This is the point where the flags must be applied, i.e., where we must apply some limited backtrack. The other strategy is to proceed in a chosen direction without a possibility to return to the choice position.

Let us analyze four possible strategies. We will do it for the general case with k coordinates, where $k \geq 3$. We will place the continuations onto a stack or into a queue, then proceed further. Those strategies, with the possibility to return to a remembered room, face another problem. In the two-dimensional case, the path from the starting point to the exit was exactly determined and there were no two or more paths possible to any room in the maze. However, with the possibility to return to a choice room, we have the possibility to reach another choice room by two or more different paths. Therefore we must memorize the choice rooms in which we have been before. This does not count for teleportation returns, but simple arrivals by a path from another room. If we arrive in a choice room r the second time by a different path, we can stop the search and ask to be teleported back to a previous choice room r', since all possible path from the room r must have been already explored, i.e., all choices for a continuation from the room r have been already placed on the stack or into the queue. When we are in dimension k for $k \geq 3$, there are $k-1$ possible continuations from a choice room. Therefore we must allow to return by teleportation to a choice room $(k-2)$-times. This is compatible with the situation in Section 1, where we allow only one teleportation return to a choice room.

A k-dimensional maze has the shape of a $n_1 \times \cdots \times n_k$ hypercube A, whose elements $A[x_1,\ldots,x_k]$ are equal to the values of a function $f\colon \mathbb{N}^k \to \mathbb{N}_{,}$ satisfying the inequality $f(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_k) < f(x_1,\ldots,x_{i-1},x_i+1,x_{i+1},\ldots,x_k)$ for each $i = 1,\ldots,k$. The starting point will be $(0,\ldots,0,n_k-1)$.

The first strategy will be completely *sequential*, presented in Algorithm 2. In each choice room r with coordinates $(x_1,\ldots,x_k)$ it will put on stack all possible continuations $(x_1,\ldots,x_{i-1},\ldots,x_i+1,x_{i+1}\ldots,x_k)$ from $i=1$ to $i=k-1$. This implies, that the continuation $(x_1,\ldots,x_{k-2},x_{k-1}+1,x_k)$ will be popped first. However, only the continuation rooms will be put on stack, which have not been visited yet. Of course, if $k=2$ then only one continuation room will be put on stack, but it will be popped and considered immediately during the next turn of the outer while-loop on Line 6. Hence, Algorithm 2 is compatible with Algorithm 1. Moreover, the use of visited rooms is superfluous for $k=2$, but introducing another if-statement would just unnecessarily complicate the algorithm. A successful path to the exit contains in the worst case at most $N = \sum_{i=1}^{k} n_i$ rooms, without counting the possible backtracks.

This version of our algorithm does not use the concept of flags, or allows to use an unbounded number of flags. The version using flags would require Algorithm 2 to be called with the parameters (f,B,F) on Line 1, where F is the number of allowed flags, followed by

Algorithm 2 Sequential Search in a k-Dimensional Maze.

Input: Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.

Output: Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function SEQ_kD_SEARCH(f, B)
2:      s ← ∅                                          ▷ Initialize stack s
3:      m ← ∅                                          ▷ Initialize memory m of visited rooms
4:      r ← (0, ..., 0, n_k − 1)                       ▷ Initialize room r
5:      s.push(r)                                      ▷ Put room r on stack s
6:      while s ≠ ∅ do                                 ▷ While stack s is nonempty
7:          r ← top(s)                                 ▷ Get room from top of stack s
8:          pop(s)                                     ▷ Pop stack s
9:          while r[j] < n_j for j = 1, ..., k − 1 & r[k] ≥ 0 do
10:             if f(r) = B then
11:                 return r
12:             else if f(r) > B then
13:                 r[k] ← r[k] − 1
14:             else if f(r) < B then
15:                 for i ← 1 to k − 1 do              ▷ For each potential continuation coordinate
16:                     r' ← r                         ▷ Copy room coordinates
17:                     r'[i] ← r'[i] + 1              ▷ Go to a neighboring room
18:                     if r'[i] < n_i & r' ∉ m then   ▷ If room within limits and not visited
19:                         s.push(r')                 ▷ Put continuation room r' on stack s
20:                         m.insert(r')               ▷ Label room r' as visited
21:                     end if
22:                 end for
23:             end if
24:         end while
25:     end while
26:     return ⊥
27: end function
```

an extension of the condition on Line 18 to "$r'[i] < n_i \ \&\ r' \notin m \ \&\ F > 0$", an introduction of the statement "$F \leftarrow F - 1$" between Lines 18 and 19, plus inserting a test "**else if** $F = 0$" with a subsequent failure command after Line 20.

The second strategy is based on a greedy heuristic to always choose first the continuation room nearest to the exit. For this, a priority queue is maintained to include the coordinates of the continuation room together with their "distance" to the exit. Since we do not know the coordinates of an exit room – which we are supposed to find – we use the difference between the label $f(x_1, \dots, x_k)$ of a continuation room r and the label B of an exit room as the priority key. This priority strategy is presented in Algorithm 3. We assume that the priority queue is implemented by a heap, therefore no code concerning an implementation of the priority queue is presented. An interested reader can find more information on priority queues and their implementation by a heap for instance in [6].

This version of our algorithm is similar to Dijkstra's shortest path algorithm [6, Section 4.4] in a graph. Dijkstra's algorithm applied directly on regular multidimensional Cartesian

Algorithm 3 Priority Search in a k-Dimensional Maze.

Input: Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.

Output: Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function PRIORITY_kD_SEARCH(f, B)
2:      q ← ∅                                          ▷ Initialize priority queue q
3:      m ← ∅                                          ▷ Initialize memory m of visited rooms
4:      r ← (0, ..., 0, n_k − 1)                       ▷ Initialize room r
5:      s.insert((r, 0))                               ▷ Insert room r into queue q with dummy key 0
6:      while q ≠ ∅ do                                 ▷ While queue q is nonempty
7:          r ← front(q).first                         ▷ Get room coordinates from front of queue q
8:          pop(q)                                     ▷ Pop queue q
9:          while r[j] < n_j for j = 1, ..., k − 1 & r[k] ≥ 0 do
10:             if f(r) = B then
11:                 return r
12:             else if f(r) > B then
13:                 r[k] ← r[k] − 1
14:             else if f(r) < B then
15:                 for i ← 1 to k − 1 do              ▷ For each potential continuation coordinate
16:                     r' ← r                         ▷ Copy room coordinates
17:                     r'[i] ← r'[i] + 1              ▷ Go to a neighboring room
18:                     if r'[i] < n_i & r' ∉ m then   ▷ If room within limits and not visited
19:                         c ← |f(r') − B|            ▷ Compute key c
20:                         q.insert((r', c))          ▷ Insert room r' with key c into queue q
21:                         m.insert(r')               ▷ Label room r' as visited
22:                     end if
23:                 end for
24:             end if
25:         end while
26:     end while
27:     return ⊥
28: end function
```

grids, where each cell represents a node and each pair of neighboring cells is connected by an edge of length 1, would potentially place each cell into the priority queue of explored nodes. Algorithm 3 places a room into the priority queue only if it matters, namely when it is a split room from where we have more than one possibility to continue. All other rooms r where $f(r) > B$ do not need to be inserted into the priority queue for the same reason as it was already mentioned in an aforementioned discussion on Algorithm 2. Moreover, we know the goal node in Dijkstra's algorithm, whereas in Algorithm 3 the exit room is unknown and must be discovered. Hence, we cannot minimize the path leading to the exit room. Therefore the distance $|f(r') - B|$ from a continuation room r' to the exit is the only value which we can minimize. This is the reason for which Algorithm 3 does not preclude backtracks, even if they are reduced to the minimum. Although in principle it is only a pseudo-problem, the use of a priority queue implies uncontrolled jumps around a maze.

The third strategy is similar to the first strategy, but instead of pushing the continuations

on stack in a fixed predefined way, it randomly permutes the sequence of continuations before placing them on stack. For this reason we call this strategy *randomized*. The strategy is implemented by a Las Vegas algorithm, therefore it always produces a correct answer. However, produced results may vary, provided that there is more than one solution, depending on the random permutation of the continuation sequence. Nevertheless, even if there is only one solution, depending on different random permutations of the continuation sequences subsequently pushed on the stack, the algorithm can follow different paths, potentially with some backtracks, to find the exit.

Algorithm 4 Randomized Search in a k-Dimensional Maze.

Input: Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.
Output: Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1: function RAND_kD_SEARCH(f, B)
2:     s ← ∅                                          ▷ Initialize stack s
3:     m ← ∅                                          ▷ Initialize memory m of visited rooms
4:     r ← (0, ..., 0, n_k − 1)                       ▷ Initialize room r
5:     s.push(r)                                      ▷ Put room r on stack s
6:     while s ≠ ∅ do                                 ▷ While stack s is nonempty
7:         r ← top(s)                                 ▷ Get room from top of stack s
8:         pop(s)                                     ▷ Pop stack s
9:         while r[j] < n_j for j = 1, ..., k − 1  &  r[k] ≥ 0 do
10:            if f(r) = B then
11:                return r
12:            else if f(r) > B then
13:                r[k] ← r[k] − 1
14:            else if f(r) < B then
15:                v ← ∅                              ▷ Initialize auxiliary vector v
16:                for i ← 1 to k − 1 do              ▷ For each potential continuation coordinate
17:                    r′ ← r                         ▷ Copy room coordinates
18:                    r′[i] ← r′[i] + 1              ▷ Go to a neighboring room
19:                    if r′[i] < n_i  &  r′ ∉ m then ▷ If room within limits and not visited
20:                        v.push(r′)                 ▷ Put continuation room r′ in vector v
21:                        m.insert(r′)               ▷ Label room r′ as visited
22:                    end if
23:                end for
24:                permute(v)                         ▷ Permute vector v uniformly at random
25:                for all r′ ∈ v do                  ▷ For the permuted sequence of continuation rooms
26:                    s.push(r′)                     ▷ Put each continuation room r′ in v on stack s
27:                end for
28:            end if
29:        end while
30:    end while
31:    return ⊥
32: end function
```

The fourth and last strategy is purely *probabilistic*. It does not store the potential

continuation rooms in a structure – a stack, a queue, or others – but it makes a probabilistic choice among possible continuations to advance, without the possibility to return back when a dead end is subsequently discovered. For this reason, this strategy is a Monte Carlo algorithm, which can produce a failure answer $\perp$ even if there exists a solution. The probability p to find an exit is equal to $E/(k-1)^S$, where S is the number of splits and E the number of exits in the maze. Potentially any room on the path from the start s to the exit can be a splitting room, therefore the (very coarse) lower bound to find an exit is equal to $E/(k-1)^N$, where $N = \sum_{k=1}^{k} n_i$ is the sum of all bounds. Recall that in the two-dimensional case ($k = 2$) there is only one exit ($E = 1$) and no splits ($S = 0$), therefore the probabilistic algorithm applied to the two-dimensional case becomes totally deterministic with the probability $p = 1$ to find the exit. This probabilistic strategy is presented in Algorithm 5.

Algorithm 5 Probabilistic Search in a k-Dimensional Maze.

Input: Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.
Output: Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function PROBA_kD_SEARCH(f, B)
2:      r ← (0, ..., 0, n_k − 1)                          ▷ Initialize room r
3:      while r[j] < n_j for j = 1, ..., k − 1  &  r[k] ≥ 0 do
4:          if f(r) = B then
5:              return r
6:          else if f(r) > B then
7:              r[k] ← r[k] − 1
8:          else if f(r) < B then
9:              i ← choose(1, ..., k − 1)          ▷ Choose a coordinate uniformly at random
10:             r[i] ← r[i] + 1                                ▷ Go to a neighboring room
11:         end if
12:     end while
13:     return ⊥
14: end function
```

4 Applications

A well-suited possibility to implement the function $f\colon \mathbb{N}^k \to \mathbb{N}$ is to use variadic polynomials over natural numbers. A variadic polynomial $p(x_1, \dots, x_k) \in \mathbb{N}[x_1, \dots, x_k]$ ensures the inequality

$$p(x_1, \dots, x_i, \dots, x_k) < p(x_1, \dots, x_i + 1, \dots, x_k) \quad \text{for each } i = 1, \dots, k,$$

since all coefficients of p are natural numbers. The bound will be $b_i = \lceil \sqrt[d_i]{B}/a_i \rceil$ for each variable x_i, where d_i is the minimum exponent of x_i and a_i is the coefficient of the monomial where the variable x_i occurs with this exponent in the polynomial p. The bounds b_i can be further reduced along the edges of the hypercube. More precisely, for each $i = 1, \dots, k$, the bound b_i can be still reduced if $p(0, \dots, 0, b_i, 0, \dots, 0) > B$ holds.

For the two-dimensional case, we can for instance use the polynomial $f(x, y) = x^3 + y^3$. In fact, this polynomial is the basis for *Taxicab numbers* [10] $\mathrm{Ta}(t)$ for $t = 1, 2, 3, \dots$ If we set the value B to one of Taxicab numbers $\mathrm{Ta}(t)$, the bounds to $m = n = \lceil \sqrt[3]{B} \rceil$, and the

starting point to $(0, \lceil \sqrt[3]{B} \rceil)$, Algorithm 1 finds a pair of values x and y whose sum of cubes is equal to $\mathrm{Ta}(t)$. An easy modification of Algorithm 1 produces *all* solutions for a Taxicab number $\mathrm{Ta}(t)$: extend the while-loop condition on Line 4 to "$x \le m-1 \ \ \& \ \ y \ge 0 \ \ \& \ \ x \le y$", replace Line 6 by "**print** (x, y)", add the instruction "$y \leftarrow y-1$" between Lines 6 and 7, and finally delete Line 13. In the same way, we can solve other problems mentioned by Silverman in [10], like the problem $x^4 + y^4 = 635318657$ solved by Euler.

For the three-dimensional case, we can use for instance the polynomial $x^3 + y^3 + z^3 = B$ with the bound $\lceil \sqrt[3]{B} \rceil$ for each variable. This problem was considered by Heath-Brown in [3], not only over natural numbers $\mathbb{N}$, but over integers $\mathbb{Z}$.

Another interesting case comes from Gauss' theorem, showing that any natural number can be written as a sum of three triangle numbers, which is equivalent to the statement that any natural number of the form $8n + 3$ can be written as a sum of squares of three odd natural numbers. This is expressed formally by $x^2 + y^2 + z^2 = 8n + 3$ for any $n \in \mathbb{N}$, where the bound for each variable is $\lceil \sqrt{8n+3} \rceil$. Hirschhorn and Sellers studied in [5] the number of *all* solutions for this problem.

An excellent example for an application is the famous Lagrange's theorem [7], showing that any natural number can be written as a sum of four squares. Formally, this can be expressed as $x^2 + y^2 + z^2 + w^2 = B$ for each $B \in \mathbb{N}$, with the bound $\lceil \sqrt{B} \rceil$ for each variable. We should mention here, that our method of generalized Young tableaux is *not* the best algorithm to compute the four squares in Lagrange's sum. More efficient algorithms exist, namely three randomized algorithms of Rabin and Shallit [9] the fastest of which has the running time $O(\log^2 B)$ provided that the Extended Riemann Hypothesis holds, their modification by Pollack and Treviño [8] with running time $O(\log^2 B / \log\log B)$, or that of Bumby [1]. It is just an interesting application for a more general setting. We can also play with Hilbert–Waring theorem [4], which says that for each natural number d there exists an associated natural number $q(d)$ such that every natural number B can be expressed as a sum of at most $q(d)$ natural numbers raised to the power d.

The coordinates of the exit in the puzzle from Section 1 are the solution of the equation $3x^{14} + 5y^{15} + 7z^{16} = 131350013988347832235$. There is only one solution to this equation.

5 Implementation and Benchmarks

All five *Generalized Young Tableaux* algorithms (respective seven if we count the modifications producing all solutions) have been implemented in `C++`. All implementations have a variant with the GNU Multiple Precision Arithmetic Library (GMP) which allows to treat numbers B of any precision. These implementations, together with data files, can be found at the `github` repository `github.com/miki-hermann/gyt`. This directory contains the individual `C++` sources, as well as the data, and a `Makefile`. There are two directories. The first one, entitled `src`, contains the `C++` sources and a `Makefile` which allows to compile the sources without typing the whole compiler command. The correspondence between the algorithms presented in this paper and their implementations is described in Table 1. The second directory, entitled `data`, contains the data files for the implemented algorithms. All implemented algorithms expect the input from `STDIN`, either typed from the keyboard after being prompted, or being piped from a file, for instance by a command like "`gyt < ../data/lagrange01.data`".

The `C++` sources have been compiled by the `g++`, version 10.2.1, with the optimization option "`-O4`". The software has been run with the benchmarks on a Dell computer with an Intel® Core™ i7-9700 CPU @ 3.00GHz × 8 processor, with 16GB RAM, running under Fedora 33. The performance of the software is quite surprizing. For instance, `gyt-2d-gmp`

Table 1 Corresondence between algorithms and C++ implementation sources.

Algorithm 1 (2-dimensional)	`gyt-2d.cpp`	`gyt-2d-gmp.cpp`
Algorithm 1 all solutions	`gyt-2d-all.cpp`	`gyt-2d-all-gmp.cpp`
Algorithm 2 (sequential)	`gyt.cpp`	`gyt-gmp.cpp`
Algorithm 2 all solutions	`gyt-all.cpp`	`gyt-all-gmp.cpp`
Algorithm 3 (priority)	`gyt-pq.cpp`	`gyt-pq-gmp.cpp`
Algorithm 4 (randomized)	`gyt-rand.cpp`	`gyt-rand-gmp.cpp`
Algorithm 5 (probabilistic)	`gyt-proba.cpp`	`gyt-proba-gmp.cpp`

computing the first solution of the 7th taxicab number (data file `taxi7.data`) takes only 31.83 seconds and `gyt-2d-all-gmp` computing *all* solutions of the same takes only 1607.15 seconds, i.e., not even 27 minutes.

The performance of the individual algorithms is measured in terms of a maximal stack or queue size, number of splits or choices, number of backtracks, and number of continuation rooms double reached. For the randomized and probabilistic versions, the most advantageous outcome out of 20 runs is presented. Table 2 summarizes the performance of the most interesting data sets. Measuring the execution time would not give a clear picture in this case, since all of them except `lagrange07.data` execute very fast. For instance, the sequential algorithm needs only 4.91 seconds on `ex01.data`, which is quite an involved data set, whereas the priority algorithm needs only 0.57 seconds to execute it, and the randomized algorithm squeezes it down even to 0.25 seconds in the best case.

6 Concluding Remarks

When we look at Table 2, we cannot decide which of the four algorithms is the clear winner. As a rule of thumb, the priority algorithm almost always outperforms the sequential algorithm. Notable exceptions are the data sets `ex01.txt` and `ex04.txt`, where the sequential algorithm pushes only 10628 or 329 rooms on stack, whereas the priority driven algorithm inserts 72910 or 213064 rooms into the queue, which indicates that the algorithm "dances" around the search space. Of course, the sequential algorithm makes in the first case more splits and backtracks, as well as it encounters more doubles. However, except for the backtracks, the priority algorithm looses in any category against the sequential version in the second case.

The priority algorithm is a clear winner for `lagrange07.data`. Both the sequential and randomized algorithms terminate with a timeout due to memory exhaustion. The priority algorithm found a solutions without backtracks and only with 16214 splits, beating even the probabilistic algorithm which does not memorize continuation rooms, but needed an incredible number of 19636633 choices. The search space must be densely populated by exits in this case, since the probabilistic algorithm almost always returns a positive anser for this data set. However, the priority algorithm looses against everybody, even against the sequential algorithm, for `ex04.data`.

The randomized and probabilistic algorithms should in principle make the same number of splits, respective choices. This is true many times, but there are cases like `lagrange09.data`, where the search space is small and populated with many solutions (there are actually 1260), but the probabilistic algorithm was not able to reach the minimum achieved by the randomized version. Note, that the priority algorithm beats everybody in this case. Neither doubles are encountered, nor backtracks are triggered.

If we do not count the probabilistic algorithm, when we wish always to receive a correct

Table 2 Performance of algorithms on chosen data sets.

Data set	measure	Algorithm 2:seq	3:priority	4:random	5:proba
`gauss-triangle4.data`	max stack/queue	49465	1858	1555	—
$x^2+y^2+z^2=$	splits/choices	51118	1857	1554	1554
2446610011	backtracks	1659	0	0	—
	doubles	2180	95152	0	—
`gauss-triangle6.data`	max stack/queue	264651	48777	16661	—
$x^2+y^2+z^2=$	splits/choices	269208	49399	16660	13024
70039266307	backtracks	4563	623	0	—
	doubles	6462	1521415	0	—
`lagrange02.data`	max stack/queue	22256	315	173	—
$x^2+y^2+z^2+w^2=$	splits/choices	11143	218	86	86
123456789	backtracks	32	0	0	—
	doubles	0	146	0	—
`lagrange07.data`	max stack/queue	timeout	32383	timeout	—
$x^2+y^2+z^2+w^2=$	splits/choices	timeout	16214	timeout	19636633
83461523083775142	backtracks	timeout	0	timeout	—
	doubles	timeout	46	timeout	—
`lagrange09.data`	max stack/queue	92	23	45	—
$x^2+y^2+z^2+w^2=$	splits/choices	46	11	22	25
2021	backtracks	2	0	0	—
	doubles	0	0	0	—
`ex01.data`	max stack/queue	10628	72910	254	—
$2x^3y^2+3y^3z^2+5z^3w^2=$	splits/choices	2058405	99665	74513	failure
84662255	backtracks	2138825	50299	101210	—
	doubles	7078935	129088	331188	—
`ex02.data`	max stack/queue	6	9	7	—
$2x^3y^2+3y^3z^2+5z^3w^2=$	splits/choices	5	5	3	3
10	backtracks	5	2	0	—
	doubles	0	0	0	—
`ex04.data`	max stack/queue	329	213064	731	—
$2x^4+3y^4+5z^4+7w^4+13u^4=$	splits/choices	6262	180091	386	failure
253930575	backtracks	14179	0	395	—
	doubles	29559	3658440	524	—
`maze01.data`	max stack/queue	16	51	27	—
$x^{10}+y^{10}+z^{10}=$	splits/choices	39	51	26	26
413575475547	backtracks	26	11	0	—
	doubles	24	22	0	—
`maze05.data`	max stack/queue	3120	130	36	—
$3x^3y^2+5y^3z^2+7x^2z^3=$	splits/choices	3555	168	35	35
29177953	backtracks	3537	44	0	—
	doubles	6875	184	0	—
`maze06.data` (using GMP)	max stack/queue	21	30	30	—
$3x^{14}+5y^{15}+7z^{16}=$	splits/choices	60	29	29	29
131350013988347832235	backtracks	45	0	0	—
	doubles	102	0	0	—

answer, the randomized algorithm beats all others in most cases. However, this is mainly due to the best performance among those 20 runs. Even if the randomized algorithm outperforms the priority one, the advantage is measured only in terms of a constant, provided we do not count the doubles. For these two algorithms, the ratio of maximal stack/queue size ranges within an interval from 1.0 to 3.6, with three notable exceptions, two in favor of the randomized algorithm (`ex01.data` and `ex04.data`) and the other in favor of the priority algorithm (`langrange02.data`). In the same spirit, the ratio of splits ranges within an interval from 0.51 to 4.8. Nevertheless, in the case of `ex04.data`, the randomized algorithm massively surpasses the priority version.

The probabilistic version works well only when the search spaces is populated with many exits. If there is only a small number of exits, namely 1 or 46, respectively, and the search space is huge, as in `ex01.data` and `ex04.data`, the probabilistic algorithm fails to find the exit. The respective randomized algorithm needed 74513 splits for the first one, but only 386 in the second. Given that there are 4 or 5 variables, respectvely, in that problem, which means the correct continuation room is chosen with probability 1/3 or 1/4, respectively, exactly 74513 or even only 386 times, the chances to find the exit by a probabilistic algorithm are practically equal to 0.

The performance of the algorithms essentially depends on the bounds. The data set `maze06.data` has a horribly big exit label B, but actually its calculated bound is relatively small: the maximal bound for `maze06.data` is 26. However, the bounds for `lagrange07.data` are all equal to 288897081.

Interested readers are invited to write their own examples and try it out with this software.

References

1 Richard T. Bumby. Sums of four squares. In David V. Chudnovsky, Gregory V. Chudnovsky, and Melvyn B. Nathanson, editors, *Number Theory: New York Seminar 1991–1995*, pages 1–8. Springer, 1996.

2 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

3 D. R. Heath-Brown. Searching for solutions of $x^3 + y^3 + z^3 = k$. In S. David, editor, *Séminaire de Théorie des Nombres, Paris, 1989–90*, volume 102 of *Progress in Mathematics*, pages 71–76. Birkhäuser, 1992.

4 David Hilbert. Beweis für die Darstellbarkeit der ganzen Zahlen durch eine feste Anzahl n-ter Potenzen (Waringsches Problem). *Mathematische Annalen*, 67:81–300, 1909.

5 Michael D. Hirschhorn and James A. Sellers. Partitions into three triangular numbers. *Australasian Journal of Combinatorics*, 30:307–318, 2004.

6 Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

7 Joseph-Louis Lagrange. Démonstration d'un théorème d'arithmétique. *Nouveaux mémoires de l'Académie royale des sciences et belles-lettres de Berlin*, 123-133, 1770.

8 Paul Pollack and Enrique Treviño. Finding the four squares in Lagrange's theorem. *Integers*, 18A:A15, 2018.

9 Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39:S239–S256, 1986.

10 Joseph H. Silverman. Taxicabs and sums of two cubes. *American Mathematical Monthly*, 100(4):331–340, 1993.